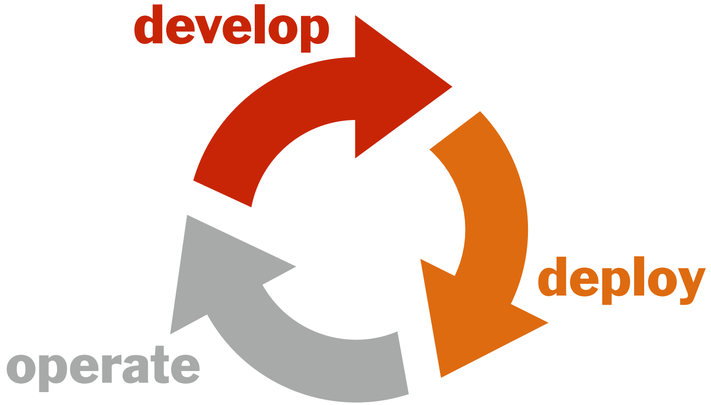
object persistence

&

transactions

learning objectives



- learn about object persistence
- learn about object serialization
- learn about object-relational mapping
- learn about transaction management

what are persistent objects?

there exists two types of memory in a computer:

- volatile memory ➡ its content disappears when the computer shuts down or restarts
- non-volatile memory ➡ its content persists when the computer shuts down or restarts

by default, objects reside in volatile memory

they are lost when the computer shuts down or restarts

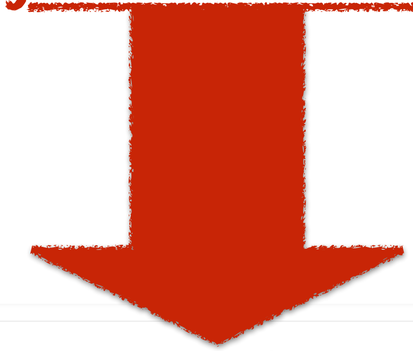
a persistent object exists primarily in non-volatile memory but also in volatile memory when it is used*

*invoked

what are persistent objects?

there exists various approaches to persist objects:

- serializing objects to a file ➡ objects are encoded as a binary stream stored into a file
- mapping objects to a database ➡ objects are mapped to a relational database
- replicating object across the network ➡ multiple replicas of each object are kept in the volatile memory of different computers, so that there always exists one replica of that object in the volatile memory of at least one computer

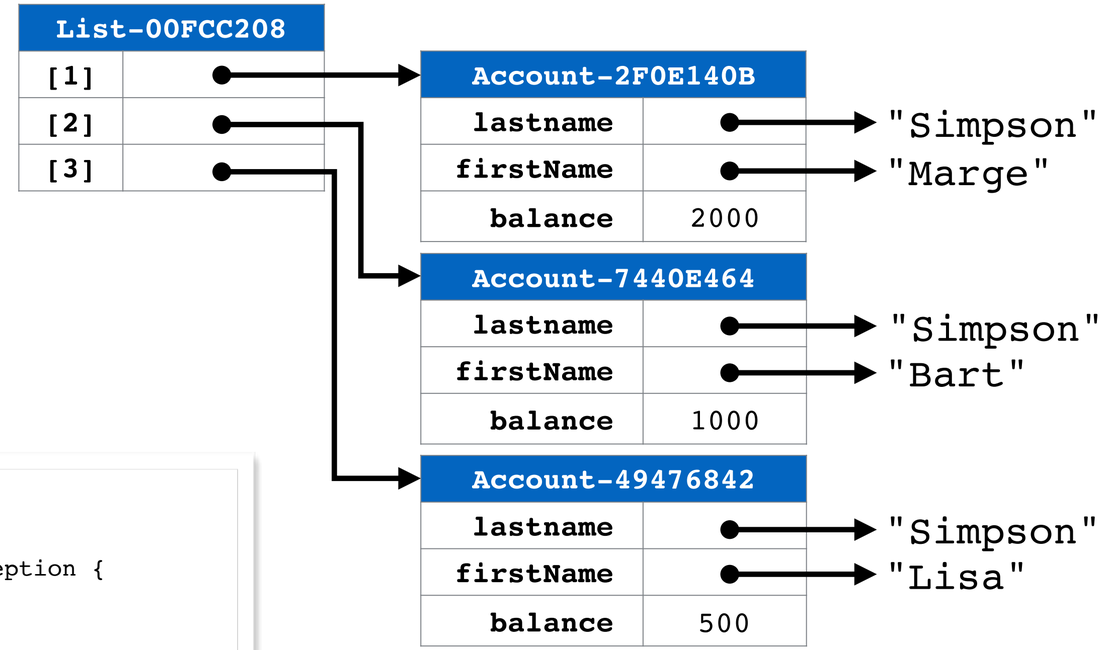


this approach comes at the cost of synchronizing the replicas of each object to keep them consistent

serializing objects to a file

```java
public class Account {
    private String lastName;
    private String firstName;
    private double balance = 0.0;

    public Account(String lastName, String firstName, double balance) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    …
    @Override
    public String toString() {
        return getClass().getSimpleName() + "-" +
        String.format("%08X", this.hashCode())  +
        "[" + lastName + ", " + firstName + ", $" + balance +"]";
    }
}
```




```java
public class Main {
    public static void main(String args[]) throws FileNotFoundException,
                                                   IOException, ClassNotFoundException {

        List<Account> accounts = List.of(new Account("Simpson", "Marge", 2000),
                                         new Account("Simpson", "Bart", 1000),
                                         new Account("Simpson", "Lisa", 500));

        System.out.println("List-" + String.format("%08X", accounts.hashCode()) + accounts);
    }
}
```

```
List-00FCC208[Account-2F0E140B[Simpson, Marge, $2000.0], Account-7440E464[Simpson, Bart, $1000.0], Account-49476842[Simpson, Lisa, $500.0]]
```

serializing objects to a file

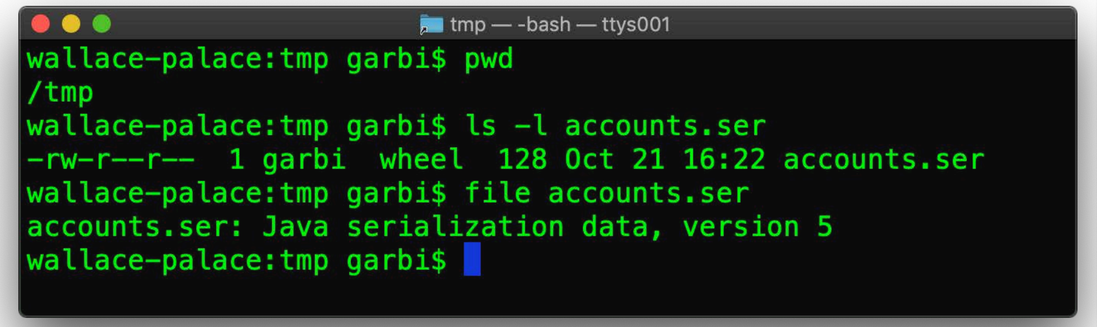
```java
import java.io.Serializable;

public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    private String lastName;
    private String firstName;
    private double balance = 0.0;

    public Account(String lastName, String firstName, double balance) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    …
    @Override
    public String toString() {
        return getClass().getSimpleName() + "-" +
        String.format("%08X", this.hashCode())  +
        "[" + lastName + ", " + firstName + ", $" + balance +"]";
    }
}
```




```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Main {
    public static void main(String args[]) throws FileNotFoundException,
                                                   IOException, ClassNotFoundException {

        List<Account> accounts = List.of(new Account("Simpson", "Marge", 2000),
                                         new Account("Simpson", "Bart", 1000),
                                         new Account("Simpson", "Lisa", 500));

        System.out.println("List-" + String.format("%08X", accounts.hashCode()) + accounts);

        FileOutputStream fos = new FileOutputStream("/tmp/accounts.ser");
        try (ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(accounts);
        }

        FileInputStream fis = new FileInputStream("/tmp/accounts.ser");
        try (ObjectInputStream ois = new ObjectInputStream(fis)) {
            accounts = (List<Account>) ois.readObject();
        }

        System.out.println("List-" + String.format("%08X", accounts.hashCode()) + accounts);
    }
}
```

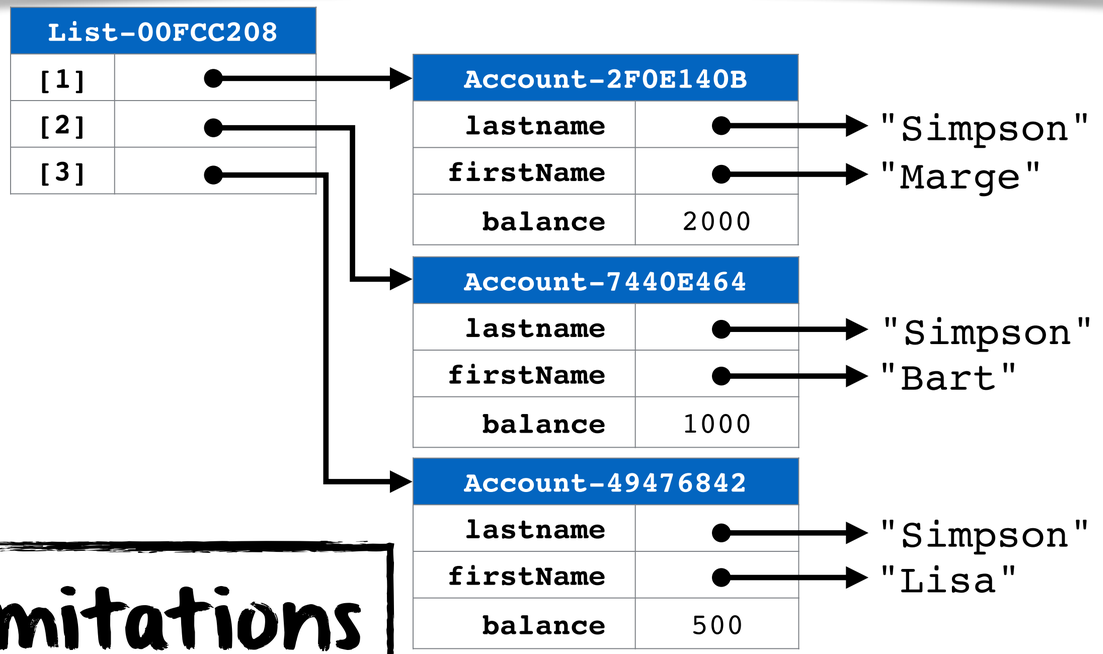
write to file

read from file

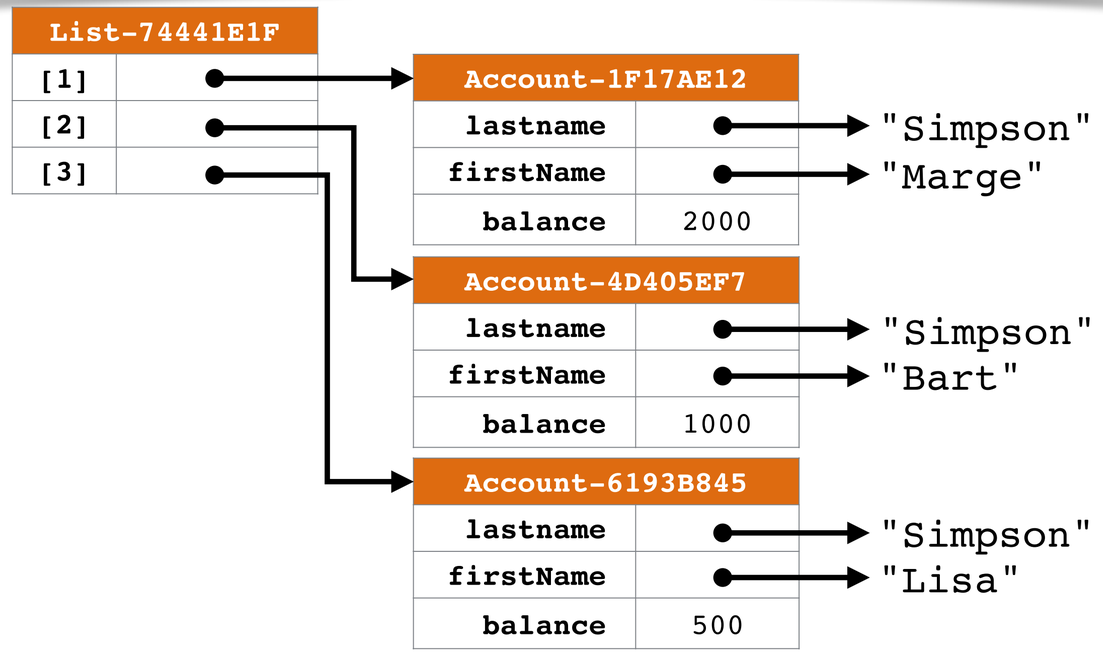
```
List-00FCC208[Account-2F0E140B[Simpson, Marge, $2000.0], Account-7440E464[Simpson, Bart, $1000.0], Account-49476842[Simpson, Lisa, $500.0]]
List-74441E1F[Account-1F17AE12[Simpson, Marge, $2000.0], Account-4D405EF7[Simpson, Bart, $1000.0], Account-6193B845[Simpson, Lisa, $500.0]]
```

serializing objects to a file

```
List-00FCC208[Account-2F0E140B[Simpson, Marge, $2000.0],
Account-7440E464[Simpson, Bart, $1000.0],
Account-49476842[Simpson, Lisa, $500.0]]
```




```
List-74441E1F[Account-1F17AE12[Simpson, Marge, $2000.0],
Account-4D405EF7[Simpson, Bart, $1000.0],
Account-6193B845[Simpson, Lisa, $500.0]]
```

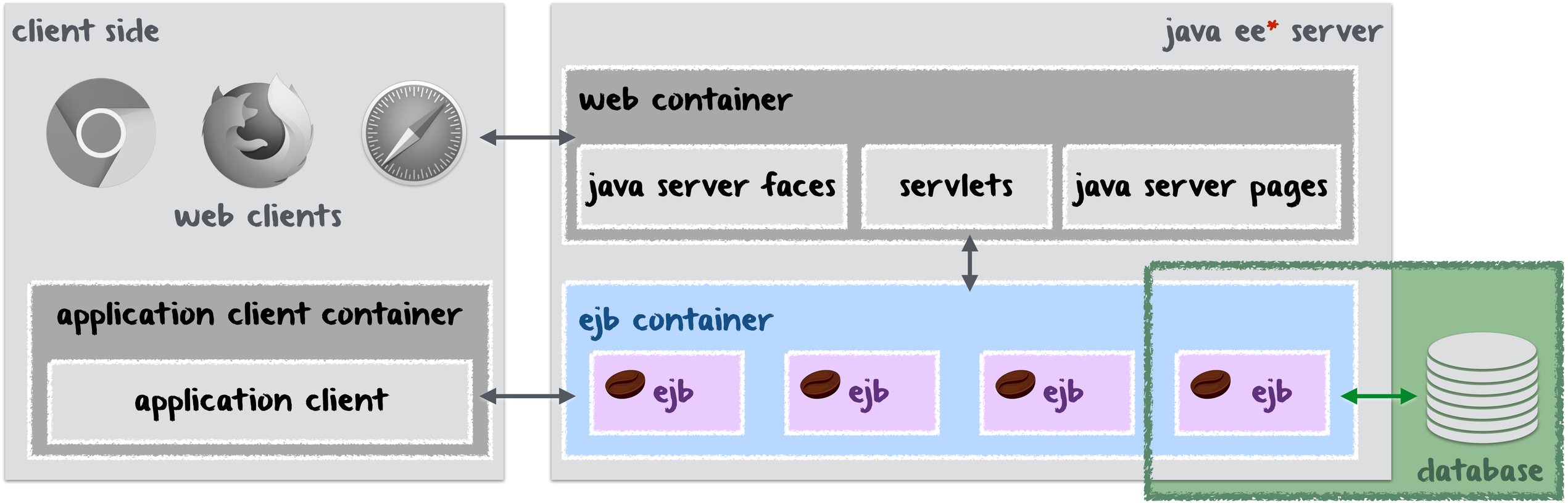


limitations

no easy navigation and querying of the serialized object graph

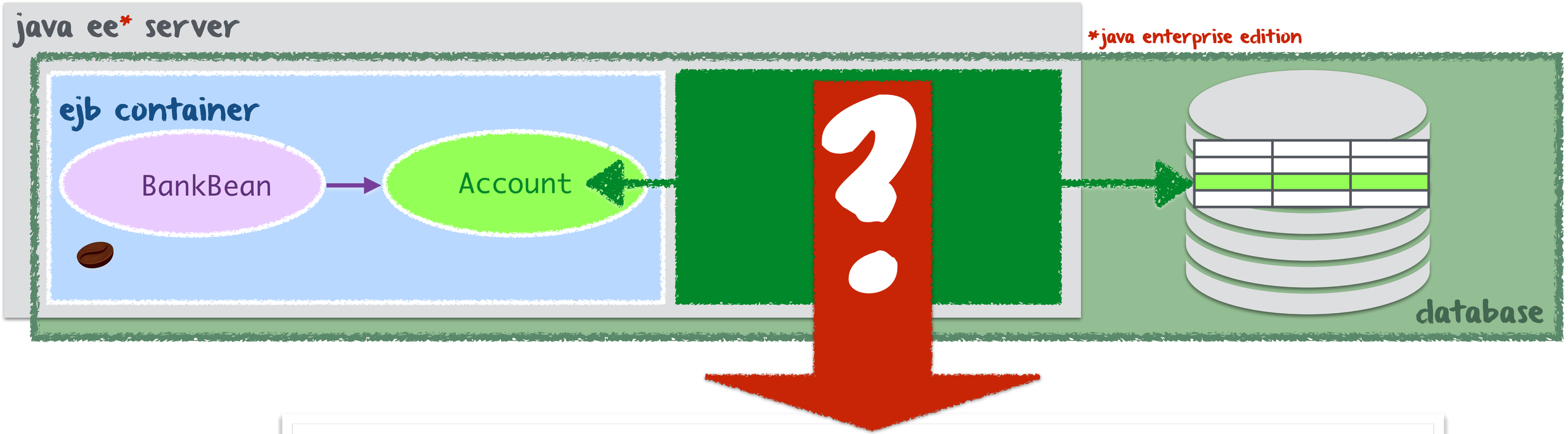
no support of legacy persistent data, stored in relational databases

mapping objects to a database



*java enterprise edition

mapping objects to a database



object-relational impedance mismatch

object-relational impedance mismatch

😕 **object identity**

- an object is uniquely identified by its implicit memory address
- a row is uniquely identified by its explicit primary key

😕 **object relations**

- an object references other objects by directly or indirectly holding references to them, i.e., by holding their implicit memory addresses
- a row references other rows via explicit foreign keys stored somewhere in the database, i.e., in the same table or in a join table

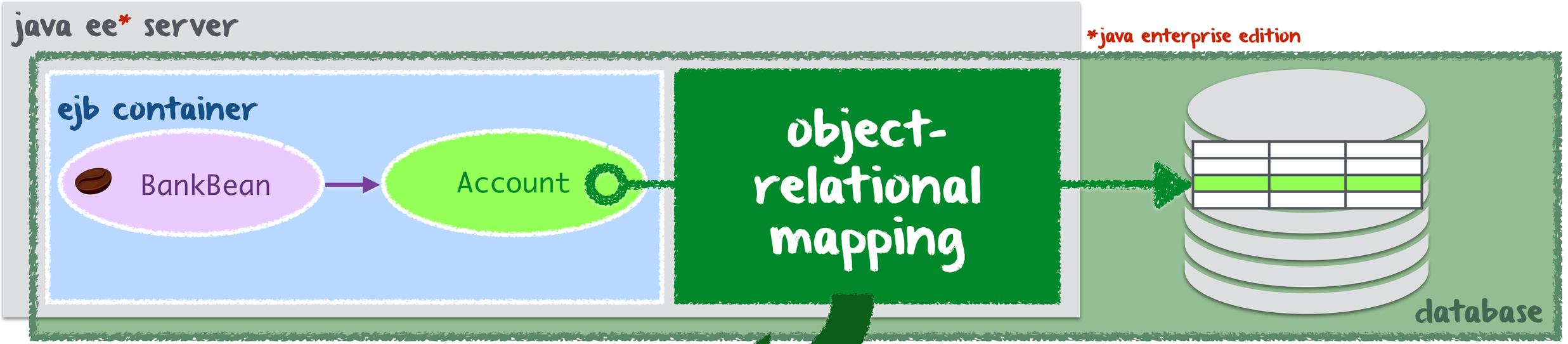
😕 **object methods**

- an object does not only encapsulate state (data), it also offers operations (methods) to manipulate that state
- a row contains data only, i.e., it comes with no set of operations to safely manipulate that data or do complex computations with it

🤔 who should be responsible for the mapping ?

which one should be the master model ? 🤔

who should be responsible for the mapping ?



the java persistence api...

... proposes declarative support for object-relational mapping

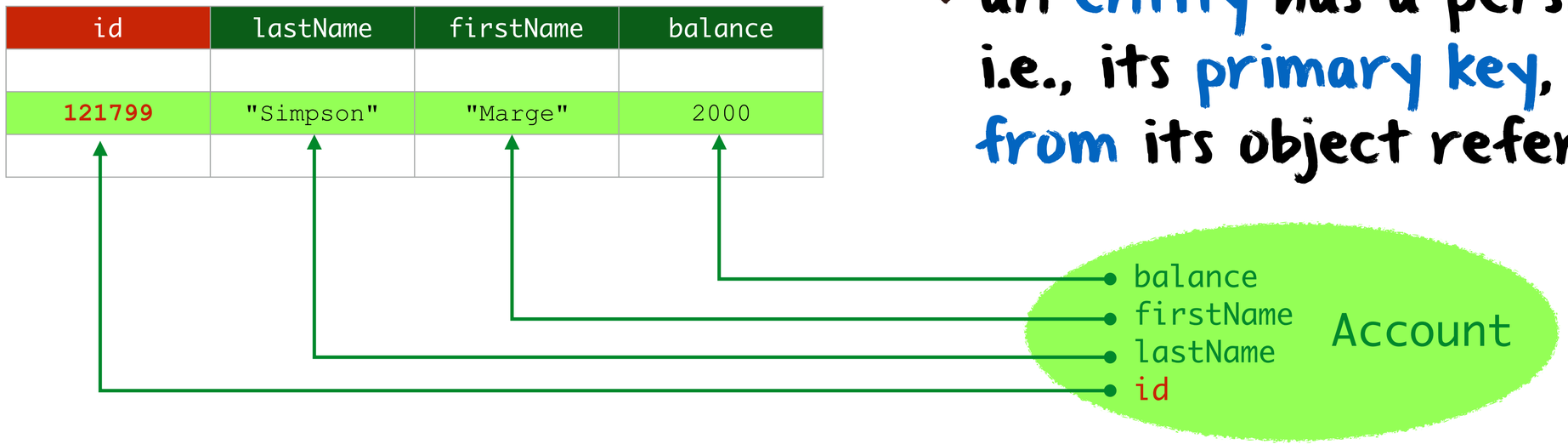
... is available for all java editions and container models

... is independent of operating systems and databases

... is based on the notion of entity

what is an entity ?

- an entity is a java object representing data from a relational database
- an entity's lifetime is independent of the application lifetime using it
- an entity has a persistent identity, i.e., its primary key, that is distinct from its object reference in memory



object identity ➧ in volatile memory, an entity is identified by its address

➧ in the database, an entity is identified by its primary key

what is an entity ?

```java
@Entity
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "BALANCE")
    private double balance = 0.0;

    public Account(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    ⋮
    public Account() {
        this.lastName = null;
        this.firstName = null;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "-" + String.format("%08X", id)
                + "[" + lastName + ", " + firstName + ", $" + balance + "]";
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" … >
  <persistence-unit name="PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.unil.doplab.persistence.Account</class>
    <class>ch.unil.doplab.persistence.Payment</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp:localhost:9092/~/databases/myh2"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

```java
    public static void persist(Object object) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("PU");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        try {
            em.persist(object);
            em.getTransaction().commit();
        } catch (Exception e) {
            e.printStackTrace();
            em.getTransaction().rollback();
        } finally {
            em.close();
        }
    }
```

```java
public class Main {
  public static void main(String args[]) {
        Account margeAccount = new Account("Simpson", "Marge");
        margeAccount.setBalance(2000.0);
        System.out.println(margeAccount);
        persist(margeAccount);
        System.out.println(margeAccount);
    }
}
```

```
Account-00000000[Simpson, Marge, $2000.0]
```

```
Account-00000001[Simpson, Marge, $2000.0]
```

account

id	lastName	firstName	balance
1	"Simpson"	"Marge"	2000

what is an entity ?

entity lookup & queries

```
public static Account findAccountByPrimaryKey(Long id) {
    Account account = null;
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("PU");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    try {
        account = em.find(Account.class, id);
        em.getTransaction().commit();
    } catch (Exception e) {
        e.printStackTrace();
        em.getTransaction().rollback();
    } finally {
        em.close();
    }
    return account;
}
```

in addition to the find-by-primary-key query, we can perform more general queries to find entities, using the Java Persistence Query Language (JP-QL)

JP-QL is similar to SQL but it manipulates objects and fields rather than rows & columns

queries can be dynamic or static

dynamic query

```
public List<Account> findAllAccounts() {
  Query query = manager.createQuery("SELECT a FROM Account a");
  return query.getResultList();
}
```

static query

```
public List<Account> findAccountsByLastName(String lastName) {
  Query query = manager.createNamedQuery("findByLastName");
  return query.setParameter("lastName", lastName).getResultList();
}
```

static queries are called named queries

```
@Entity
@NamedQueries({
        @NamedQuery(name = "findByAccountID",  query = "SELECT a FROM Account a WHERE a.id = :id"),
        @NamedQuery(name = "findByLastName",   query = "SELECT a FROM Account a WHERE a.lastName = :lastName"),
        @NamedQuery(name = "findByBalance",    query = "SELECT a FROM Account a WHERE a.balance = :balance")})
public class Account implements Serializable {
⋮
}
```

what is an entity ?

object relations ➡ an object references other objects by directly or indirectly holding references to them, i.e., by holding their implicit memory addresses

➡ a row references other rows via explicit foreign keys stored somewhere in the database, i.e., in the same table or in a join table

```java
@Entity
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "BALANCE")
    private double balance = 0.0;

    @OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private List<Payment> payments;

    public void addPayment(Payment payment) {
        payment.setAccount(this);
        payments.add(payment);
    }
    ⋮
}
```

```java
@Entity
public class Payment implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "EXECDATE")
    private Date execDate;

    @Column(name = "AMOUNT")
    private double amount;

    @ManyToOne
    private Account account;
    ⋮
}
```

account

id	lastName	firstName	balance
1	"Simpson"	"Marge"	2000

payment

id	amount	execdate	account_id
4	150.0	2019-01-02	1
5	300.0	2019-02-04	1
6	450.0	2019-03-06	1
7	600.0	2019-04-08	1

account_payment

account_id	payment_id
1	4
1	5
1	6
1	7

object-relational impedance mismatch

who should be responsible for the mapping ?

➡ Java Persistence API (JPA)

object identity

- in volatile memory, an entity is identified by its address
- in the database, an entity is identified by its primary key

object relations

- in volatile memory, an entity references other entities by directly or indirectly holding references to them, as for any java object
- in the database, an entity references other entities via explicit foreign keys stored in the same table and/or in a separate join table

object identity and relations are expressed and managed automatically and transparently by the Java Persistence API (JPA)

object-relational impedance mismatch

object methods ➡ an object does not only encapsulate state (data), it also offers operations (methods) to manipulate that state

➡ a row contains data only, i.e., it comes with no set of operations to properly manipulate that data or do complex computations with it

↳ deep mismatch ➡ no mapping possible

which one should be the master model ?

the relational model ?

the object model ?

what do you think?

entity lifecycle callbacks

```
@Entity
@Table(name = "ACCOUNT")
public class Account {
    @PrePersist
    void prePersist() { ... }

    @PostPersist
    void postPersist()  { ... }

    @PreRemove
    void preRemove()  { ... }

     ⋮

     ⋮

    @PostRemove
    void postRemove() { ... }

    @PreUpdate
    void preUpdate()  { ... }

    @PostUpdate
    void postUpdate() { ... }

    @PostLoad
    void postLoad() { ... }
}
```

entity states & transactions

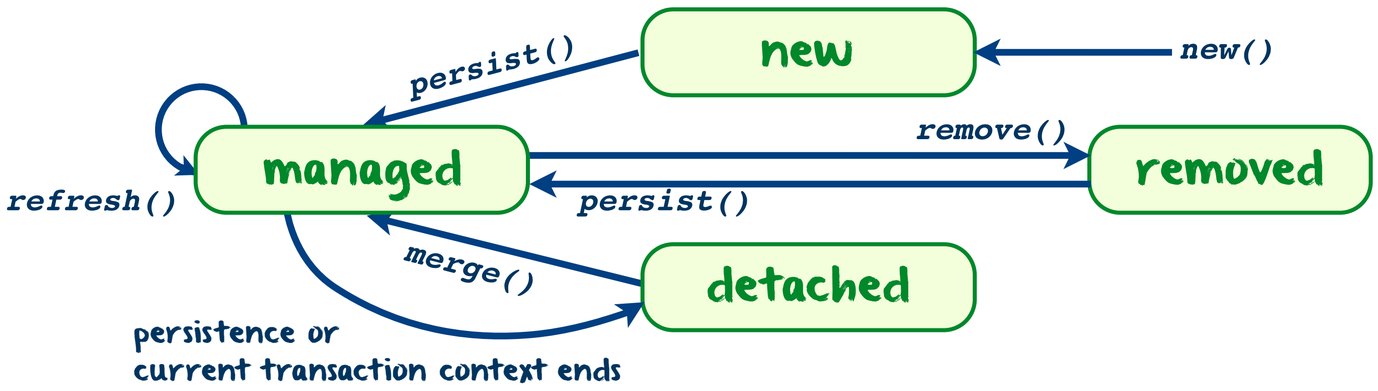
after a `manager.persist(account)` or a `manager.merge(account)` call, the account entity is scheduled for being synchronized (written) to the database

the entity will actually be written when the current transaction commits

until then, the entity is in managed state

an entity is only kept in sync with the database when in managed state

an entity is only in managed state when manipulated in the context of a transaction



new	just created but not yet bound to a persistent identity in the database
managed	bound to a persistent identity and scheduled to be synchronized with the database
detached	bound to a persistent identity but not synchronized with the database
removed	bound to a persistent context and scheduled for removal from the database

using entities from enterprise beans

```java
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class BankBean implements BankBeanLocal {

    @PersistenceContext
    private EntityManager manager;

    @Override
    public Account openAccount(String lastName, String firstName) {
        Account account = new Account(lastName, firstName);
        manager.persist(account);
        return account;
    }

    @Override
    public void closeAccount(long accountID) {
        Account account = manager.find(Account.class, accountID);
        manager.remove(account);
    }

    @Override
    public double deposit(long accountID, double amount) {
        Account account = manager.find(Account.class, accountID);
        return account.deposit(amount);
    }

    @Override
    public Account findAccount(long accountID) {
        return manager.find(Account.class, accountID);
    }
    ⋮
}
```

dependency injection

why do we have to find the entity in every method?

```java
@Stateful
public class AccountBean {
    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    private EntityManager manager;

    private Account account = null;

    public void open(int accountNumber) {
        account = manager.find(Account.class, accountNumber);
        if (account == null) {
            account = new Account();
            manager.persist(account);
        }
    }
    public void deposit(int amount) {
        if (account == null) throw new IllegalStateException();
        account.deposit(amount);
    }
    public String getLastName() {
        if (account == null) throw new IllegalStateException();
        return account.getLastName();
    }
    ⋮
}
```

the session facade pattern consists in having a stateful session bean act as front-end for an entity

atomic transactions

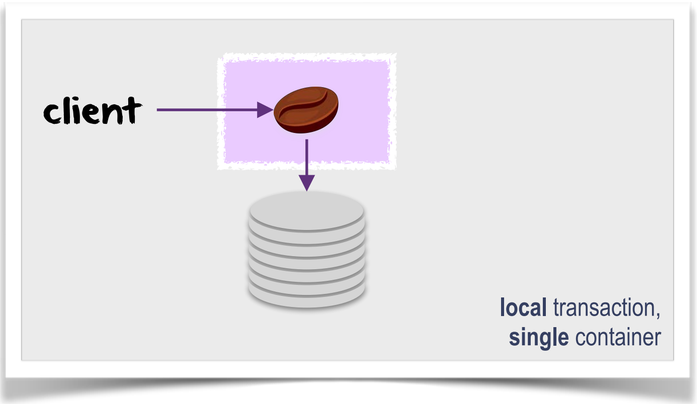
a transaction T ensures the four acid properties:

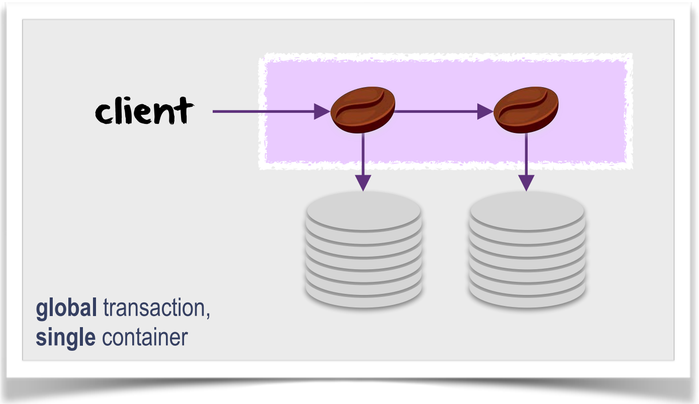
- **a**tomicity — T appears either committed or aborted with respect to failures
- **c**onsistency — T does not compromise the consistency of the data it manipulates
- **i**solation — T appears indivisible with respect to all other transactions
- **d**urability — T being committed, its effects will survive subsequent crashes

this is ensured by persistence

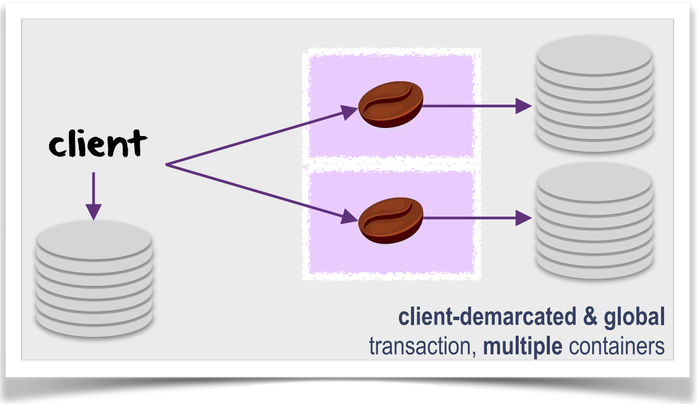
transactions with ejbs

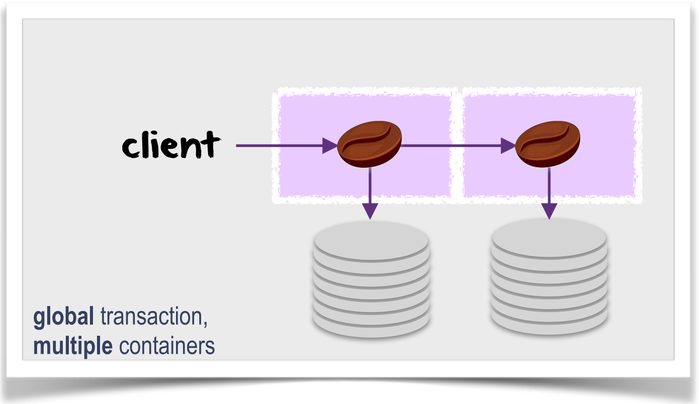
the ejb transactional model supports various scenarios





the ejb transactional model offers two ways to express transactions





programmatically, via bean-managed transactions

declaratively, via container-managed transactions

programmatic transactions

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class WarehouseBean implements SessionBean {
    private DataSource ds;
    private Connection cn;
  @Resource SessionContext ctx;
    public void ship(String productId, String orderId, int quantity) {
        try {
            ds = (javax.sql.DataSource) ctx.lookup("jdbc/EmployeeAppDB");
            cn = ds.getConnection();
            cn.setAutoCommit(false);
            updateOrderItem(productId, orderId);
            updateInventory(productId, quantity);
            cn.commit();
        } catch (Exception ex) {
            try {
                cn.rollback();
                throw new EJBException("Transaction failed: " + ex.getMessage());
            } catch (SQLException sqx) {
                throw new EJBException("Rollback failed: " + sqx.getMessage());
            }
        } finally {
            cn.close();
        }
     ⋮
    }
```

local transaction

global transaction

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.BEAN)
public class TellerBean implements TellerRemote {
     ⋮
    public void withdrawCash(double amount) {
        UserTransaction ut = context.getUserTransaction();
        try {
            ut.begin();
            updateChecking(amount);
            machineBalance -= amount;
            insertMachine(machineBalance);
            ut.commit();
        } catch (Exception ex) {
            try {
                ut.rollback();
            } catch (SystemException syex) {
                throw new Exception("Rollback failed: " + syex.getMessage());
            }
            throw new Exception("Transaction failed: " + ex.getMessage());
        }
    }
}
```

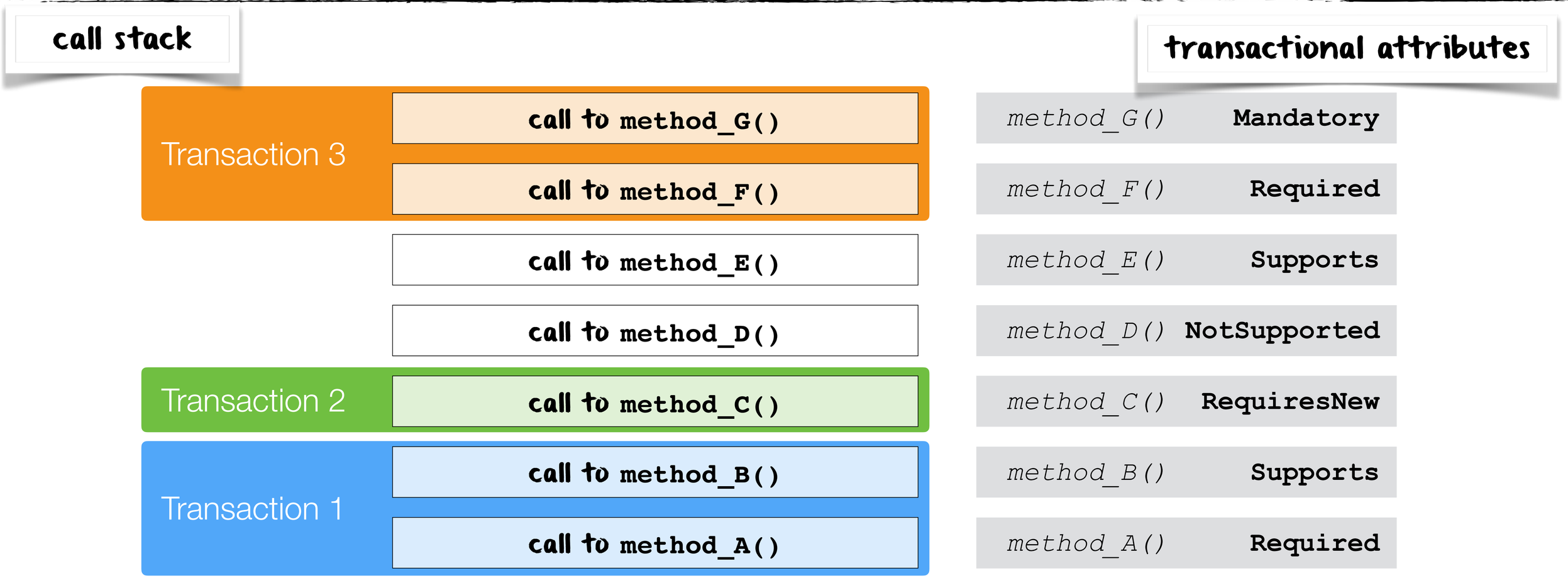
declarative transactions

a transactional attribute can be associated with each method via annotations

attribute	meaning
`NotSupported`	if a client's transaction exists, it is suspended
`Supports`	if a client's transaction exists, it is continued
`Required`	if a client's transaction exists, it is continued, otherwise, the container starts a new transaction
`RequiresNew`	the container always starts a new transactions, if a client's transaction exists, it is suspended first
`Mandatory`	the client must be in a transaction when calling
`Never`	the client must not be in a transaction when calling

declarative transactions

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class AccountBean implements AccountLocal {
     ⋮
   @TransactionAttribute(javax.ejb.TransactionAttributeType.SUPPORTS)
    public double getBalance() { ... }
}
```



declarative transactions

how to tell the container to rollback a transaction, because of some applicative problem occurred?

```
    @Resource
    SessionContext context;
```

```
public void transferToSaving(double amount) throws InsufficientBalanceException {
        checkingBalance -= amount;
        savingBalance += amount;

        if (checkingBalance < 0.00) {
             context.setRollbackOnly();
             throw new InsufficientBalanceException();
        }

        updateChecking(checkingBalance);
        updateSaving(savingBalance);
    ⋮
}
```